Pointers

Lecture #10

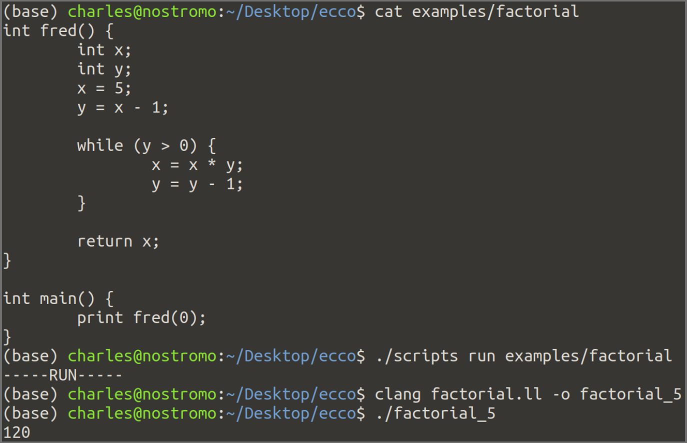
Charles Averill

Introduction to Compiler Design
The University of Texas at Dallas

Fall 2023

The Current State of the Compiler

Previously, we added support for function calls:



```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/factorial
int fred() {
        int x;
        int y;
        x = 5;
        y = x - 1;

        while (y > 0) {
                x = x * y;
                y = y - 1;
        }

        return x;
}

int main() {
        print fred(0);
}
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/factorial
-----RUN-----
(base) charles@nostromo:~/Desktop/ecco$ clang factorial.ll -o factorial_5
(base) charles@nostromo:~/Desktop/ecco$ ./factorial_5
120
```

Goal

We want to add support for pointers:

```
int main() {
    int a; int b;
    int c; int d;
    int *e; int *f;

    a = 18; print a;                   // 18
    e = &a; b = *e; print b;           // 18

    c = 12; print c;                   // 12
    f = &c; d = *f; print d;           // 12

    int g; int *h; int **j; int ***k;
    g = 5; h = &g; j = &h; k = &j;
    int n; int *m; int **l;
    l = *k; m = *l; n = *m;
    print n;                           //  5
}
```

Goal

This is a really cool feature. In the future it'll let us pass more versatile objects into functions. We'll be able to call `malloc` when we support the C standard library.

However, to accomplish this, we will have to refactor most of our number system. This will be annoying!

Plan

1. Add ampersand token, dereference meta-token
2. Update our Number, LLVMValue types to store pointer information
3. Update type parsing to check for pointers
4. Update our expression parsing to handle unary pointer operators
5. Update our register loading code
6. Add generator functions for addressing and dereferencing variables

New Tokens

Adding an ampersand token is no problem. However, we have an issue with our derefernce token.

Initially, I just wanted to reuse the STAR token. The "times" operator and dereference operator are both stars, so why not?

New Tokens

Answer: We're building an ABSTRACT syntax tree. A concrete syntax tree *would* use a STAR token for both.

Abstract syntax trees encode **ideas** about the program, not specific details about the ASCII representation of the code itself.

Because of this, I've added a new DEREFERENCE token. We'll see in a bit how we use the two tokens in tandem.

Updating Number and LLVMValue

Our Number and LLVMValue classes store information about data flowing through our program. Right now they can really just handle numbers. We need to update them so they can recognize whether they're storing pointers or not.

ACWJ accomplishes this by adding types INTPTR, CHARPTR, etc. on top of the INT and CHAR types. Why is this suboptimal?

My first approach to solving this problem included storing a ”stores_pointer” bool field to the Number and LLVMValue classes. Why is this suboptimal?

Updating Number and LLVMValue

Solution: we store a "pointer_depth" int field in both classes. Regular ints and chars will have a pointer depth of 0, int pointers depth 1, int pointer pointers depth 2, etc.

This ends up generalizing really well. We can update the pointer depth field by addition or subtraction when we address or dereference a variable to create our new Numbers/LLVMValues.

Type Parsing

Utilizing our new tokens and our Number updates, we can now parse pointer variable declarations.

I've added a `match_type` function that matches either an INT or a CHAR token, then tries to parse as many stars as possible. It then stores those two data points into a Number object and returns it. We use this in `declaration_statement` and will eventually use it in `function_declaration`.

Parsing unary operators in expressions

To add an operator, normally we would just add entries to our precedence table. However, we also have to do some static semantic checking to make sure programmers can't use the pointer operators illegally.

If we added standard unary operator support, you could do

```
int x;
int y = ***x;
int z = &&&y;
```

as unary operators generally operate on expressions, and are expressions themselves.

Let's look at `prefix_operator_passthrough`.

Register Loading Update

Previously, we used a list of registers to determine what data was loaded and what wasn't. This won't work any longer, why not?

Register Loading Update

Previously, we used a list of registers to determine what data was loaded and what wasn't. This won't work any longer, why not?

Answer: with variable-pointer-depth data, knowing whether data is loaded or not is not explicitly dependent on the pointer depth any more, it's instead dependent on what an operation expects. LLVM won't let you add two `int*`s, so we know that when we add two numbers, the input LLVMValues should be loaded such that they have a pointer depth of 0.

The solution is to add a new input to `llvm_ensure_registers_loaded` that specifies the expected pointer depth of what is considered "loaded".

New Generator Functions

For getting the address of a variable, we can simply allocate space for a new register on the stack, then

`store <var type> @<var name>, <type with ptr depth + 1> %<new reg>`.

Dereferencing is accomplished through the `load` instruction that we're already familiar with.